UNCLASSIFIED

CURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
a SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
4. NAME OF PERFORMING ORGANIZATION: Administrative Sciences Department, Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): AS	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

1. TITLE (Include Security Classification)

AN INTRODUCTION TO X WINDOW APPLICATION DEVELOPMENT (U)

2. PERSONAL AUTHOR(S)

Rust, David M.

3a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 10/90 TO 03/92	1992, March, 23	70

6. SUPPLEMENTARY NOTATION: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

7. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Computer Programming, Graphical User Interface, portable systems, toolkits, widget, window, windowing system, Xlib, X Window

9. ABSTRACT (Continue on reverse if necessary and identify by block number)

The challenge to developing applications for computer-based windowing systems is generating code for the graphical interface elements. Each windowing system offers its own set of protocols for building the graphical units, but these protocols are rarely portable across different hardware platforms. The X Window System transcends many of these incompatibilities and offers a standard for creating graphics. It is operating system and network independent. However, the basic programming library for X Window offers little sophistication for an application's graphical interface development. Higher level tools make up for the shortcomings of the generic X Window System.

This thesis converts an Expert System Knowledge Acquisition and Policy Evaluation program using Cognitive Feedback (ESKAPE/CF) from the SunView windowing system to X Window. The new application, called XESKAPE/CF, contains the same functionality as the original program even though the migration from SunView to X Window required an extensive reworking of the program's interface code. The thesis also extends the basic X Window library of functions with more advanced objects. These objects offer additional functionality to the XESKAPE/CF application's interface.

0. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
[X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	UNCLASSIFIED	
2a. NAME OF RESPONSIBLE INDIVIDUAL: Ishore Sengupta	22b. TELEPHONE (Include Area Code): (408) 646-3212	22c. OFFICE SYMBOL: AS/Se

D FORM 1473, 84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Approved for public release; distribution is unlimited

An Introduction to X Window Application Development

by

David Michael Rust
Lieutenant, United States Navy
B.S., Rice University, 1986

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
March 1992

ABSTRACT

The challenge to developing applications for computer-based windowing systems is generating code for the graphical interface elements. Each windowing system offers its own set of protocols for building the graphical units, but these protocols are rarely portable across different hardware platforms. The X Window System transcends many of these incompatibilities and offers a standard for creating graphics. It is operating system and network independent. However, the basic programming library for X Window offers little sophistication for an application's graphical interface development. Higher level tools make up for the shortcomings of the generic X Window System.

This thesis converts an Expert System Knowledge Acquisition and Policy Evaluation program using Cognitive Feedback (ESKAPE/CF) from the SunView windowing system to X Window. The new application, called XESKAPE/CF, contains the same functionality as the original program even though the migration from SunView to X Window required an extensive reworking of the program's interface code. The thesis also extends the basic X Window library of functions with more advanced objects. These objects offer additional functionality to the XESKAPE/CF application's interface.

TABLE OF CONTENTS

I. INTRODUCTION 1

A. THESIS OBJECTIVES AND SCOPE 1

B. BACKGROUND 2

C. ORGANIZATION OF THE STUDY 2

D. DISCLAIMER 3

II. X WINDOW PROGRAMMING LAYERS 4

A. PROGRAMMING IN X WINDOW 4

1. Widgets 4

2. Toolkit Hierarchy 4

B. XLIB: THE X WINDOW LIBRARY 5

C. MIDDLE LEVEL TOOKITS 5

1. Intrinsics 6

a. Xt: The X Toolkit 6

b. Xaw: The Athena Widget Set 6

2. XView 6

D. HIGH LEVEL TOOLKITS 7

1. Open Look Intrinsics Toolkit (OLIT) 7

2. Motif 7

III. AN INTRODUCTION TO XLIB PROGRAMMING TECHNIQUES 8

A. GETTING STARTED 9

1. Connecting the Client to the Server 9

2. Linking to a Screen 9

3. Loading Fonts 10

B. USING WINDOWS 10

1. Creating Windows 10

2. The Graphics Context 12

3. Mapping Windows 12

4. Drawing Inside the Window 13

C. EVENT LOOP PROCESSING 13

1. Handling User Input 14

2. Refreshing the Screen 15

IV. XESKAPE/CF: THE X WINDOW VERSION OF ESKAPE/CF 16

A. FUNCTIONAL COMPARISON TO ESKAPE/CF 16

B. MOTIVATION FOR THE PROGRAM CONVERSION 19

C. INITIAL APPROACH TO THE PROGRAM CONVERSION 19

D. MAIN PROGRAM CONTROL: THE EVENT LOOP 20

E. BUILDING WIDGETS 23

1. Interactive Text 23

a. Creating the Widget 25

b. Tying the Widget to the Event Loop 26

c. Using the Widget In the XESKAPE/CF Program 32

2. Static Text 35

3. Buttons 37

V. CONCLUSIONS AND RECOMMENDATIONS 39

A. ENVIRONMENTAL CONSIDERATIONS 39

1. Portability 39

2. Interface Consistency....39
B. APPLICATION CONSIDERATIONS....40
1. Application Size....40
2. Programming Experience....40
APPENDIX A: THE HISTORY OF X WINDOW....41
APPENDIX B: THE STRUCTURE OF X WINDOW....43
APPENDIX C: THE GETTEXT WIDGET....47
APPENDIX D: GLOSSARY OF TERMS....57
LIST OF REFERENCES....60
BIBLIOGRAPHY....62
INITIAL DISTRIBUTION LIST....63

I. INTRODUCTION

A. THESIS OBJECTIVES AND SCOPE

The emergence of windowing systems on computer workstation environments introduces entirely new aspects to application development. These graphic-intensive systems burden developers because of the complexities of programming onto bit-mapped screens (Brown 1989). Graphical applications are usually linked heavily to a particular hardware platform and operating system. Therefore, porting such applications to multiple platforms becomes costly in both time and programming effort.

The main purpose of this thesis is to transplant James Conner's **E**xpert **S**ystem **K**nowledge **A**cquisition and **P**olicy **E**valuation tool using **C**ognitive **F**eedback, **ESKAPE/CF** (1991) program from the SunView windowing system to X Window. The rationale of this conversion is to promote a more portable operational environment for the application. The new program, named **XESKAPE/CF** for **X** Window version of **ESKAPE/CF**, contains the same functionality as the original program, however the migration from SunView to X Window required an extensive reworking of the program's interface code. The goal of achieving maximum portability within X Window itself required the use of only low-level X routines. This restriction forced the construction of several interface objects to mimic those used in the original ESKAPE/CF application.

The scope of this thesis is limited to an evaluation of the low-level X Window programming required for construction of the XESKAPE/CF program interface. Additionally the thesis describes the locally generated extensions to the basic X Window library.

B. BACKGROUND

Bitmapped graphical displays have displaced the character display terminal as the mainstream interface tool for computer workstations (Lainhart 1991). This movement toward the graphical interface attempts to transcend the barriers of incompatible hardware platforms and operating systems. In addition, the desire for application consistency and portability has driven systems toward graphical displays (Clanton 1991).

In spite of much variance in operating systems and network topologies, the X Window system overcomes many traditional compatibility issues and offers a truly standardized windowing system. It gains strength from its network and operating system independence. X Window is portable across hardware, software and network systems (Brown 1989). In recent years it has become a standard for workstation windowing systems (Thareja and Ramachandran 1991). As a programming environment, generic X Window offers little except low-level graphics routines. Nevertheless, numerous higher level development tools have been built to extend the windowing environment.

C. ORGANIZATION OF THE STUDY

Beyond the introduction and conclusion, this thesis consists of three sections. Chapter II describes the different programming levels associated with X Window and some of the development tools available for the system. Chapter III introduces the more critical aspects of low-level programming within X Window. It serves as a brief tutorial on building a basic X Window program.

Chapter IV describes the motivation and methodology used to convert the ESKAPE/CF program into the XESKAPE/CF application. The extensions to the basic X Window library are explained and examples of their incorporation within XESKAPE/CF are provided.

Appendix A contains a brief history of the X Window system. Such knowledge provides the reader with the background and motivation behind the windowing system. Appendix B describes the structure of X Window and how it relates to the hardware platforms on which it operates. Source code providing a sample extension to the basic X Window library is listed in Appendix C. Appendix D presents a glossary of terms related to windowing systems and X Window in particular.

D. DISCLAIMER

While much effort has been spent testing the XESKAPE/CF program and its associated code, no guarantees are implied or made. Operating or modifying the application will be done at the user's own risk. The author hopes that the XESKAPE/CF program serves as an effective and productive tool. Any comments or inquiries concerning the program's source code should be directed to the author or the thesis advisor.

II. X WINDOW PROGRAMMING LAYERS

A. PROGRAMMING IN X WINDOW

The X Window System contains several levels of tools to aid in program development. Near the bottom, these tools retain the advantage of X Window's standardization and portability. As a programmer moves higher in tool sophistication, the more proprietary their work becomes.

1. Widgets

The graphical user interface concept opens the user to a number of graphical objects that can be displayed and manipulated on the screen. These objects can be windows, buttons, scroll panels and sliders. The common name for these data structures is widget. Any object that can be created and manipulated fits the definition (Johnson and Reichard 1991). Widgets are reusable and can be uniquely altered for the individual implementation (Lainhart 1991).

2. Toolkit Hierarchy

There are three distinct levels to the X Window hierarchy of toolkits. Figure 1 shows the relationship of these levels within the X Window environment. In its most

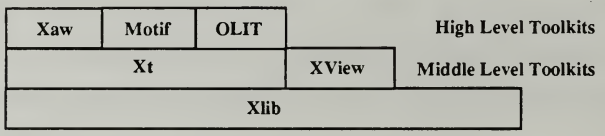

Figure 1. The X Window Toolkit Hierarchy

generic form X Window contains only a low-level set of library routines called Xlib. These functions allow access to the system's graphics and interface functions (Reichard and Johnson 1991). Sitting on top of the Xlib library are the X Toolkit Intrinsics. This layer

provides support for generic widgets as well as the ability to create more customized ones. Toolkits residing at this intermediate level are middle level toolkits. The top-level toolkits are known as widget sets. They are robust widget sets, providing powerful, sophisticated objects to the programmer.

B. XLIB: THE X WINDOW LIBRARY

Xlib is a set of C programming language routines that link directly to the X server. It represents the closest thing to assembly language in the X Window System (Yee 1991). Xlib protects the programmer from having to worry about the details of connecting to an X server and maintaining a network link (Reichard and Johnson 1991). Containing over 300 routines, Xlib offers functions that create, move, resize and destroy windows; select fonts; draw text and graphics in color, gray scale and monochrome; and recognize user input from the keyboard and mouse.

However, no sophisticated objects exist in Xlib. The library offers no widgets and the programmer must handle details associated only with the graphical interface. All graphical objects must be created from the basic building block of Xlib, the window. This lack of sophistication does not imply that Xlib fails as a useful programming tool. On the contrary, it serves as the basis for all higher level intrinsics and widgets sets. The portability of X Window applications comes directly from the standardization of these Xlib routines.

C. MIDDLE LEVEL TOOKITS

In order to remove some of the abstraction from the programmer, middle-level toolkits have been built above the Xlib level. While numerous toolkits at this level exist, significant examples include the X Consortium's Intrinsics and Sun's XView.

1. Intrinsics

The most widely known and utilized toolkit is the Intrinsics. The X Consortium sets the standard for the Intrinsics and imposes its inclusion in X Window implementations. (Lainhart 1991)

a. Xt: The X Toolkit

The Intrinsics consists of two parts. The first, Xt, is a set of routines built on top of Xlib to facilitate program design (Lainhart 1991). Xt does not contain widgets. It helps the programmer create them. Many higher level toolkits use the Xt intrinsic as their interface to the Xlib library.

b. Xaw: The Athena Widget Set

While part of the Intrinsics, the Xaw toolkit or Athena Widget Set is actually a high level toolkit. It uses Xt as its foundation to establish widgets and contains the major objects desired in a widget set. Additionally, the Athena Widget set is the X Consortium standard toolkit.

2. XView

Sun Microsystems offers three toolkits for use in the X Window environment: the OPEN LOOK Intrinsics Toolkit (OLIT), XView and the NeWS Toolkit (TNT). Only XView uses Xlib as its sole basis and therefore qualifies as a middle-level toolkit. TNT is fairly new and uses the PostScript language for client/server communications. It will not be discussed in this thesis. (Millikin 1990)

The XView Toolkit was designed to aid in the porting of existing SunView applications to the X Window environment. Sun attempted to retain as much of the graphical interface as possible but still had to abandon the SunView Pixrect library in favor of the X Window Xlib graphics routines. Unfortunately the task of converting an application from Sun Windows (using SunView) to X Window (using XView) is not as

simple as the process might appear. Differences in object modeling and imaging account for some of the difficulties. (Millikin 1990)

D. HIGH LEVEL TOOLKITS

The high-level toolkits provide the programmer with the most sophistication in widgets while adhering to a particular graphical interface appearance. The two most widely used toolkits are Unix Software Laboratories' Open Look Intrinsics Toolkit (OLIT) and Open System Foundation's Motif.

1. Open Look Intrinsics Toolkit (OLIT)

OLIT is the toolkit for the Open Look environment. It utilizes the Xt intrinsics to define its interface components included in the Open Look widget set. The resultant functionality of applications built on the OLIT is tied heavily to Open Look's graphical interpretation of the X Window environment. While portability to other window managers is possible, OLIT's reliance on the Open Look widget set locks an application within the entire Open Look/ Open Windows concept.

2. Motif

Motif can be described as a window manager, but that portion is simply an addition. It is really a full-fledged toolkit. Rapidly becoming a strong contender for the X Window standard, Motif offers a very attractive three-dimensional, sculptured style to its windows and widgets. However, functionally Motif does not differ much from OLIT. (Padovano 1991)

III. AN INTRODUCTION TO XLIB PROGRAMMING TECHNIQUES

Unlike more traditional programming using a third generation language, programming using the X Window library (Xlib) presents some unanticipated challenges. The existence of only windows within the Xlib toolkit forces a programmer to become more creative if sophisticated structures are desired. A window need not be limited to just a viewing area. It can be designed into a push button. In turn the push button can become part of a scroll panel. Windows can serve as the foundation as well as the building blocks of many objects in an Xlib program.

The X Window library offers the necessary functions to create almost any application. Most of the approximately 300 commands are simple function calls for use within the C programming language. The programmer need only initialize the proper variables and then include them as parameters of the Xlib calls. However, the Xlib functions present the programmer with only basic building blocks. Windows can be created, moved, resized and destroyed. Graphics and text can be drawn inside the windows. Fonts can be loaded and icons created.

The comparison by Yee (1991) of Xlib to assembly language demonstrates the scope within which a programmer must work. In order to create an Xlib program that prints a simple message onto the screen the following steps must occur:

a. Establish a connection with the X server and one of its screens
b. Load a font for the text
c. Create the window in which to place the text
d. Create a graphics context with which to draw the text
e. Map the window onto the screen
f. Make the window visible on the screen
g. Draw the text in the window
h. Continually handle events pertaining to this program

While much of this process appears intimidating, a programmer can still produce powerful and very streamlined applications using only the Xlib commands.

A. GETTING STARTED

Xlib first requires the initialization of several link and structures before the first window can be created. Most of these procedures exist because of the network-oriented nature of X Window.

1. Connecting the Client to the Server

The client/server relation within X Window necessitates establishing a connection between the application program and the X server. The Xlib command *XOpenDisplay* links the client to the server.

```
Display*display;
char*display_name = NULL;

display = XOpenDisplay (display_name);
```

While not a particularly difficult task, the programmer should be aware that Xlib does not handle errors in setting up this connection. Placing the *XOpenDisplay* command inside an *if* statement along with an appropriate message will notify the user of potential problems with establishing this critical connection. (Nye 1990)

2. Linking to a Screen

A server can be further subdivided into screens. The screen refers to the physical display device, usually a CRT. The client must know how to identify the screen onto which it will ultimately display windows and data. A path to the desired screen is obtained with the *DefaultScreen* command.

```
Display     *display;
int         screen;

screen = DefaultScreen( display );
```

Most servers will only control a single screen, but some will connected to two or more. Screen identification is necessary when creating a window hierarchy.

3. Loading Fonts

A font in X Window is a series of bitmaps representing the shapes of a character set (Nye 1990). These shapes may be text, symbols or shapes. Two methods exist to identify fonts in Xlib. One method uses a font ID number while the other method uses a pointer to a font structure. The *XLoadFont* command returns a font ID number.

```
Display     *display;
char        *font_name;
Font        font_id;

font_id = XLoadFont( display, font_name );
```

The *XLoadQueryFont* command returns a pointer to a font's *XFontStruct.* This structure contains sizing information for the font.

```
XFontStruct     *font_structure;

font_structure = XLoadQueryFont( display, font_name );
```

Before any font resource can be accessed by an application it must be loaded. Font libraries containing a multitude of different font styles and sizes exist for X Window systems. The standard, default fonts named "fixed" and "variable" are available in nearly every system.

B. USING WINDOWS

Xlib provides full control over windows created in the X Window environment. While toolkits may offer more robust objects than just windows, Xlib offers control over minute details associated with windows. Basic Xlib window operations involve window creation, mapping and utilization.

1. Creating Windows

X windows exist in a hierarchy. Each window has a parent window and may have one or more windows as either siblings or offspring. All windows attached to a screen are

the ultimate descendants of a special window called the root window. Covering a screen's entire background, the root window provides a common parent for an application's main windows (Johnson and Reichard 1990).

In order to define the root window structure the programmer must utilize the server link and screen identifier. The *RootWindow* command returns the value representing the top level window.

```
Display         *display;
int             screen;
Window          parent_window;

parent_window = RootWindow( display, screen );
```

The Xlib commands *XCreateSimpleWindow* and *XCreateWindow* both initialize windows below the root window level. The level of detail regarding window parameters is the main difference between the commands. *XCreateSimpleWindow* inherits many window attributes from the parent window while *XCreateWindow* forces the programmer to explicitly define them.

```
Display         *display;
Window          parent_window,
                window;
int             x, y;
unsigned int    width, height, border_width;
unsigned long   border_pixel_type, background_pixel_type;

window = XCreateSimpleWindow( display, parent_window,
                              x, y, width, height, border_width,
                              border_pixel_type, background_pixel_type );
```

Through these Xlib calls the programmer establishes such attributes as window location, size and parent. Additionally, each window has a multitude of other attributes that determine unique aspects of the window's appearance and function. The X Window programmer maintains full control over windows via Xlib functions accessing these attributes.

2. The Graphics Context

If a window is to contain any data at all it will most likely require the use of one or more Xlib graphics primitives to draw the data. The graphics context (GC) is a resource that determines the appearance of all graphics inside a window except the border and background (Nye 1990). A graphics context is established by the Xlib *XCreateGC* command and can be modified by several commands that affect individual GC attributes.

```
Display         *display;
Window          window;
unsigned long   gc_attribute_mask;
XGCValues       gc_attribute_values;
GC              graphics_context;

graphics_context = XCreateGC( display, window,
                              gc_attribute_mask,
                              &gc_attribute_values );
```

Graphic contexts serve to minimize the communications between client and server. The GC resource's information resides in the server after being initially sent from the client. The server will display all graphics according to the graphic context's style. Traffic between the client and server is reduced since the client need not transmit graphical style information for each call to a graphics primitive.

3. Mapping Windows

After the program has created a window, it must make an Xlib call to map the window onto the screen. Mapping a window tells the server that the window is ready to be drawn onto the screen. The Xlib command *XMapWindow* notifies the server of the program's intention.

```
Display         *display;
Window          window;

XMapWindow( display, window );
```

Windows that have been mapped will not become visible until after the server's event queue has been emptied. Whether or not the window is obscured by another application's window is a concern for the window manager. However, the programmer must control the stacking order of overlapping windows within an application.

4. Drawing Inside the Window

With the window visible and a GC assigned to it, all graphics primitives in Xlib are ready for use. Being primitives, the tools do not offer much beyond drawing text, points, lines, arcs, circles, ellipses and rectangles (Nye 1990). Nevertheless, a programmer can create more sophisticated graphics from any or all of these basic tools.

Most of the graphics primitives require only the destination window, location and sizing values for the item to be drawn. For example, the *XDrawRectangle* command draws a rectangle in the designated window. Other Xlib commands follow a similar pattern.

```
Display         *display;
Window          window;
GC              gc;
int             x, y;
unsigned int    width, height;

XDrawRectangle( display, window, gc, x, y, width, height );
```

C. EVENT LOOP PROCESSING

X Window programs are event-driven. An event is an asynchronous notification from the server to the client of particular actions such as input from the keyboard or exposure of a window. Events are only sent to a client if the client specifically requested notification of the event's occurrence (Nye 1990). Unlike other types of programming, event-driven programs are not sequential. They do not wait for input at a particular point in the program code. Instead, an X Window application cycles continually through an event loop waiting

for messages from the server that certain actions have occurred. The client will then respond to these events if the program includes code to handle them.

A common structure for an event loop is a while loop with no terminating condition. Within this loop a *switch* structure based on event type will direct program flow to the appropriate action. (Nye 1990)

```
Display         *display;
XEvent          event;

while ( 1==1)
{
    XNextEvent( display, event );
    switch( event.type )
    {
        case Expose:
            /* Perform action for Expose event */
            break;
        case ButtonPress:
            /* Perform action for ButtonPress event */
            break;
        default:
            /* Perform default action */
            break;
    }
}
```

Since every program should have a graceful means of terminating, one of the actions should lead to an exit routine.

1. Handling User Input

Events resulting from direct user action include *ButtonPress*, *ButtonRelease* and *KeyPress* events. After one of these events is recognized, the event loop directs the program flow to routines tailored to respond to the input. First, the program must determine from which window an event originated. Knowing the window localizes where the input occurred. A *ButtonPress* event from a window representing a push button will initiate the button's notify procedure. Likewise, *KeyPress* events, which indicate keyboard input, must

be queried to extract the character value entered. Appropriate action will then result based upon the character entered.

2. Refreshing the Screen

The *Expose* event for a window results from either the initial mapping of a window or the revealing of all or part of it by another window. Such an event should command that the window's contents be redrawn. Most window managers will redraw the window's border but will do nothing to restore any graphics previously drawn inside that border. Therefore the event loop determines the window associated with the event and then calls a procedure that redraws its contents.

IV. XESKAPE/CF: THE X WINDOW VERSION OF ESKAPE/CF

The X Window version of the Expert System Knowledge Acquisition and Policy Evaluation tool using Cognitive Feedback (XESKAPE/CF) serves to extract policy knowledge from an expert. As with the original application called ESKAPE/CF, it reduces the time required for a knowledge engineer to capture expertise for an expert system (Conner 1991).

A. FUNCTIONAL COMPARISON TO ESKAPE/CF

The ESKAPE/CF program operates on the SunView interface. XESKAPE/CF is designed for X Window. As knowledge acquisition programs, the XESKAPE/CF and ESKAPE/CF applications are identical. They use the same data structure for storing the expert knowledge, and files generated by one program are fully compatible with those of the other. The majority of the code affecting knowledge representation, storage and manipulation remains unaltered following the conversion to X Window.

The XESKAPE/CF program also does little to modify the interface of the original ESKAPE/CF program. Figure 2 shows the main screen for the ESKAPE/CF program. In Figure 3 the X Window version, XESKAPE/CF, demonstrates the functional similarity between the two applications. The subtle differences in object appearance results from the manner in which these objects are drawn by each windowing system. A button in one program may look different than a button in the other application; however, functionally the buttons initiate the same action. This characteristic holds true for many of the other interface objects.

Figure 2. Main ESKAPE/CF Screen

Figure 3. Main XESKAPE/CF Screen

B. MOTIVATION FOR THE PROGRAM CONVERSION

The desire to extend the portability of the ESKAPE/CF program motivated the conversion from SunView to X Window. SunView is a proprietary windowing system and does not have the widespread base of the X Window system. Porting the application to X Window provides a larger spectrum of users to verify the application's validity.

Two main goals drove the methodology of converting ESKAPE/CF to XESKAPE/CF. The first goal required XESKAPE/CF to remain compatible with the SunView version. The program data files had to be fully portable as a minimum standard. Ideally, the program's data structures and numerical methods would be left unchanged in XESKAPE/CF. This functional level of compatibility would further enhance the portability of the ESKAPE/CF concept.

The second conversion goal mandated that XESKAPE/CF operate independently of any vendor's toolkit. This restriction limited the scope of application conversion to Xlib since Xlib programs operate on any standard X Window system. By programming XESKAPE/CF at the Xlib level, full X Window portability could be realized without the need for building multiple versions for the different toolkits.

C. INITIAL APPROACH TO THE PROGRAM CONVERSION

The initial approach to the program conversion intended to replace the SunView interface code with comparable Xlib syntax. Automating the conversion process was even considered. However, major differences between the SunView program structure and that required for X Window forced all but a total adandonment of the main program routine and organization.

SunView is a kernel-based system and does not correspond well to X Window's event-driven structure. The conversion of ESKAPE/CF to XESKAPE/CF would require a complete reprogramming of the application's interface code. The SunView library of

functions contains much similarity with high-level X Window toolkits. It is a complete widget set. The ESKAPE/CF application's extensive use of widgets forced development of a simple collection of X Window interface objects.

D. MAIN PROGRAM CONTROL: THE EVENT LOOP

The main control for the ESKAPE/CF program arises from the SunView command *window_main_loop()*. The application invokes this command for the base window of the program while the details behind it are hidden within the SunView library and need not be of concern for the programmer. All SunView objects are pre-linked to this main loop command.

The XESKAPE/CF application also uses a main loop to drive program control. While more advanced toolkit libraries may offer this function as a single-line procedure, Xlib forces the programmer to develop a complete event loop. The following listing shows the basic structure to XESKAPE/CF's main program control structure and the event loop procedure. The function called *EventLoop()* responds to all requested events individually and manages all locally defined widgets.

```
Display    *display;          /* the server connection */
Window     base_window;       /* the main program window */

/*------------------------- Loop forever looking for events */

while ( 1 == 1 )
    {
    EventLoop( display, base_window );
    }
```

```
void
EventLoop( display, window )
Display     *display;              /* the server connection */
Window      window;
{
    XEvent          event;

    /*------------------------- Block on input, awaiting an event from X --------*/
    XNextEvent( display, &event );
    /*----------------- Check if the event was handled by the button interface ---*/
    if ( ButtonEvent( display, &event ) == True )
            {
            return();
            }
    /*----------------- Check if the event was handled by a GetText structure ----*/
    if ( GetTextEvent( display, &event ) == True )
            {
            return();
            }
    /*----------------- Check if event was handled by DisplayText structure ----*/
    if ( DisplayTextEvent( display, &event ) == True )
            {
            return();
            }
    /*----------------- Check if it was an Expose for scroll panel's view window */
    if ( RefreshScrollPanel( display, &event ) == True )
            {
            return();
            }
    /*----------------- Check if the event involves the slider for correlation ----*/
    if ( slider_event( display, &event ) == True )
            {
            return;
            }
```

```
        /*----------------- Decode event and call a specific routine to handle it ------*/
        switch( event.type )
                {
                case ButtonPress:
                        /*-------Handle cue selection from scroll panel's pixmap --*/
                                PointerOnCue( display, cue_panel, event,
                                                cue_locations );
                        break;
                case LeaveNotify:
                        /*-------Set input focus to root window --*/
                        if (event.xcrossing.window == base_window )
                                XSetInputFocus( display, root_window,
                                                RevertToPointerRoot,
                                                CurrentTime );
                        break;

                case ResizeRequest:
                        /*-------Do not allow resizing the base window --*/
                        if (event.xresizerequest.window == base_window )
                                XResizeWindow( display, base_window,
                                                SCREEN_WIDTH,
                                                SCREEN_HEIGHT );
                        break;

                default:
                        break;
                }
        return();
}       /*------------------ end EventLoop ---------------------------------*/
```

At the core of the main control procedure is the endless *while* loop, waiting continuously for events from the server. A quit button within the application will invoke a termination procedure to stop the program. Nested within the *while* loop, the *EventLoop()* function is invoked. Within this event loop, a series of *if* statements check if an event refers to one of the locally defined widgets. The commands linking these widgets to the *EventLoop()* are discussed later in this chapter. The *switch* structure handles any remaining events. These events are not associated with any of the widgets. They refer to such tasks as resizing and redirecting the input focus of the program's main window.

E. BUILDING WIDGETS

The SunView environment supports objects such as panels, text, menus, scroll bars and sliders. The Xlib library does not offer this flexibility. Xlib provides no sophisticated objects beyond the window. However, a GUI-based program such as XESKAPE/CF requires metaphors such as interactive text, push buttons and scroll panels. These widgets had to be built from scratch using Xlib.

The XESKAPE/CF program prompted the construction of several widgets. Push buttons are the most commonly used widgets. Scroll panels, text entry panels and text display panels were also utilized. All of these widgets had to be fabricated from the Xlib commands since none of them preexisted. The interactive text entry panel called the *GetText* widget will be used as the principal example. The fundamental structure of the other widgets remains similar.

1. Interactive Text

Even simple text entry from the keyboard becomes a major obstacle in the Xlib environment. The capability to build such a widget exists but the widget structures themselves do not. The *GetText* widget obtains input from the keyboard and echoes the input onto the screen. Appendix C includes the necessary C code to implement the *GetText* widget. This section refers to this Xlib code to explain the construction and use of this simple widget.

Widgets must be constructed upon some type of data structure. This structure can be as simple as an array of data elements or a series of elements linked by pointers. The simplest method involves the use of arrays. Each element of the array contains a list of elements that describe the individual attributes of the widget. Most widgets are windows with added functionality so they are identified by their window identifier. Other attributes include parent window, location, size, font type, graphics context and labels.

The *GetText* widget includes all of the attributes mentioned in the above paragraph. The widget's structure is defined as:

```
typedef   struct
          {
          Display          *display;      /* the server connection */
          Window           window;        /* the id of the widget's window */
          Window           parent;        /* the id of widget's parent window */
          GC               gc;            /* graphics context for the widget */
          unsigned long    fore, back;    /* fore & background settings */
          int              string_length;
          int              (*function) (); /* notify procedure for widget */
          char             string[SIZE];  /* string to display in the widget */
          }
          GetTextStruct;
```

The current value of the string being obtained from the keyboard must be stored so that it can be displayed onto the screen. The *string* element of the widget's structure hold this value.

Next, functions to control the widget's creation and operation were designed. Most of these procedures exists only for the widget and will become extensions to the library of local functions. The XESKAPE/CF program has only to call them as required. These functions provide the *GetText* widget with its consistent look and feel. Functions associated with the operations of the *GetText* widget include *CreateGetText()*, *SetGetText()*, *GetTextEvent()*, *AdvanceGetText()* and *RedrawGetText()*.

a. Creating the Widget

Prior to use, the widget must be created in a similar fashion as creating windows. The *GetText* widget uses a separate function called *CreateGetText()* to initialize the new instance of a widget.

```
Window
CreateGetText( display, parent, x, y, width, height,
                    fore, back, font_id, string_length, function )

Display         *display;
Window          parent;
int             x, y, width, height;
unsigned long   fore, back;
Font            font_id;
int             string_length;
int             (*function)();

{
    Window      w;
    GC          gc;

    /*---------------------- Find a slot */

    if ( get_text_widgets_used < ( MAX_GET_TEXT_WIDGETS - 1 ) )
            {
            w = CreateWindow( display, parent,
                        x, y, width, height, 0,
                        fore, back,
                        ExposureMask | ButtonPressMask | KeyPressMask |
                        EnterWindowMask | LeaveWindowMask );

            /*---------------- Create a GC and assign font */

            gc = MakeGC( display, w, fore, back );
            XSetFont( display, gc, font_id );

            /*---------------- Store values */

            GetText[ get_text_widgets_used ].display = display;
            GetText[ get_text_widgets_used ].window = w;
            GetText[ get_text_widgets_used ].parent = parent;
            GetText[ get_text_widgets_used ].gc = gc;
```

```
            GetText[ get_text_widgets_used ].fore = fore;
            GetText[ get_text_widgets_used ].back = back;
            GetText[ get_text_widgets_used ].function = function;
            GetText[ get_text_widgets_used ].string_length = string_length;
            strcpy( GetText[ get_text_widgets_used ].string, NULL_STRING );

            XFlush( display );
            /*------------------ Increment slot pointer */

            get_text_widgets_used++;

            return( w );
            }
        else
            return( 0 );
}
/* --- end CreateGetText ---*/
```

First a new window is created using the *CreateWindow()* function. This function simplifies the Xlib process of window initialization. The code for the function is included in Appendix C. The new window forms the foundation of the widget. It has no border and is therefore invisible when mapped to the screen. Next a graphics context is set up for drawing the text. Again a function to simply the procedure is used. Code for the *MakeGC()* function is also in Appendix C. The font desired for the text is linked to the graphics context and finally the object's parameters are saved into the widget's data structure for later use.

Among the parameters is the widget's notify function. This function is the procedure that the widget will call after an Enter, Line Feed or Return key is pressed. The widget's string value at that point is passed as a parameter to the notify function.

b. Tying the Widget to the Event Loop

The link between widget and event loop is the most critical aspect of widget construction. A well designed widget will respond to most every event in a logical manner. The response should anticipate the result. For example, a program receiving an *Expose*

event on a widget's main window should presume that the widget needs to be redrawn and initiate a redraw function. If all widgets thoroughly manage themselves, a program's main event loop can consist mainly of calls to the widgets' event handling routines.

The *GetText* widget uses the function *GetTextEvent()* to handle the arrival of an event associated with itself. Events of concern are the *Expose* event, *KeyPress* event, *EnterNotify* event, *LeaveNotify* event and *ButtonPress* event.

```
boolean
GetTextEvent( display, event )
Display          *display;
XEvent           *event;
{
    int          which_widget;

    switch ( event->type )
        {
        case Expose:
            which_widget = FindGetTextWidget(display,
                                        event->xexpose.window);

            if ( which_widget >= 0 )
                {
                RedrawGetText( display, which_widget );
                return ( True );
                }
            break;
        case KeyPress:
            which_widget = FindGetTextWidget( display,
                                        event->xkey.window );

            if ( which_widget >= 0 )
                {
                GetChar( display, event, which_widget );
                return ( True );
                }
            break;
```

```
            case EnterNotify:
                which_widget = FindGetTextWidget( display,
                                        event->xcrossing.window );

                if ( which_widget >= 0 )
                    {
                    AdvanceGetText( display,
                                        event->xcrossing.window );
                    return ( True );
                    }
                break;
            case LeaveNotify:
                which_widget = FindGetTextWidget( display,
                                        event->xcrossing.window );

                if ( which_widget >= 0 )
                    {
                    XClearWindow( display,
                                        event->xcrossing.window );
                    RedrawGetText( display, which_widget );
                    return ( True );
                    }
                break;
            case ButtonPress:
                which_widget = FindGetTextWidget( display,
                                        event->xbutton.window );

                if ( which_widget >= 0 )
                    {
                    AdvanceGetText( display, event->xbutton.window );
                    return ( True );
                    }
                break;
            default:
                break;
            }

        XFlush( display );

        return ( False );
    }
    /*--- end GetTextEvent ---*/
```

The *Expose* event indicates a need to redraw the current state of the widget onto the screen. This event will arrive after the widget has been mapped to the screen or after it has been uncovered by another window thereby requiring it be redrawn. If the *Expose* event originated from a *GetText* widget, then the object must be redrawn and the *RedrawGetText()* function is called.

```
void
RedrawGetText( display, which_get_text )
Display     *display;
int         which_get_text;
{
    int             font_height;

    font_height = font_struct->ascent + font_struct->descent;

    XDrawString( display, GetText[ which_get_text ].window,
                 GetText[ which_get_text ].gc, 2, font_height,
                 GetText[ which_get_text ].string,
                 strlen( GetText[ which_get_text ].string ) );
    return;
}
/*--- end RedrawGetText ---*/
```

The *KeyPress* event arises after any keystroke. As with the *Expose* event, the *GetTextEvent()* function includes code to verify each *KeyPress* event with the *GetText*

widgets to determine if action is required. The *GetChar()* routine from Nye (1990) determines the appropriate action based on the character entered.

```
void
GetChar( display, event, which_get_text )
Display         *display;
XEvent          *event;
int             which_get_text;     /* the GetText struct to operate on */

{
    int                 count;
    char                *buffer[1];
    int                 bufsize = BUFSIZE;
    KeySym              keysym;
    XComposeStatus      compose;
    int                 length;

    count = XLookupString( event, buffer, bufsize, &keysym, &compose );

    /*---------------------- if Enter, Return or LineFeed call the GetText function */

    if (( keysym == XK_Return ) || ( keysym == XK_KP_Enter ) ||
                                        ( keysym == XK_Linefeed ))
        {
        XClearWindow( display, GetText[ which_get_text ].window );

        RedrawGetText( display, which_get_text );

        GetTextExec( display, which_get_text );

        return;
        }
```

```
    /*---------- if a regular key, add it to the string unless > string_length */

    else if ((( keysym >= XK_KP_Space ) && ( keysym <= XK_KP_9 )) ||
             (( keysym >= XK_space ) && ( keysym <= XK_asciitilde )))
            {
            if (( strlen( GetText[ which_get_text ].string ) + strlen( buffer )) >=
                                    GetText[ which_get_text ].string_length )
                    XBell( display, 100 );
            else
                    strcat( GetText[ which_get_text ].string, buffer );
            }

    /*---------- skip if key is a modifier key */

    else if (( keysym >= XK_Shift_L ) && ( keysym <= XK_Hyper_R ))
            ;

    /*---------- if Backspace or Delete, remove one char */

    else if (( keysym == XK_BackSpace ) || ( keysym == XK_Delete ))
            {
            if ( strlen( GetText[ which_get_text ].string ) > 0 )
                    {
                    length = strlen( GetText[ which_get_text ].string );
                    GetText[ which_get_text ].string[ length - 1 ] = NULL;
                    XClearWindow( GetText[which_get_text].display,
                                            GetText[which_get_text].window );
                    }
            else
                    XBell( display, 100 );
            }

    /*---------- if any other key, skip it and beep */

    else
            XBell( display, 100 );

    UpdateActiveGetText( display, which_get_text );

    return;
    }
/*--- end GetChar ---*/
```

The *EnterNotify* and *ButtonPress* events are indicators that the widget has been selected to accept character input. The program directs the focus of the keyboard to the widget's window following one of these events. Changing the keyboard focus will direct all subsequent *KeyPress* events into the indicated window. The function *AdvanceGetText()* changes the keyboard focus.

```
void
AdvanceGetText( display, get_text_widget )
Display         *display;
Window          get_text_widget;
{
    int         which_get_text;


    which_get_text = FindGetTextWidget( display, get_text_widget );

    UpdateActiveGetText( display, which_get_text );

    XSetInputFocus( display, GetText[ which_get_text ].window,
                RevertToParent, CurrentTime );

    return;
}
/*--- end AdvanceGetText ---*/
```

The *GetText* widget holding the keyboard focus contains a caret or cursor at the text input location. The function *UpdateActiveGetText()* draws the caret inside the widget's window.

The *LeaveNotify* event indicates that the widget is no longer selected for keyboard input. The caret is removed from the widget's window and the function *RedrawGetText()* redraws the string without the caret.

c. Using the Widget In the XESKAPE/CF Program

Once the *GetText* widget has been created, it can be mapped to the screen and used to extract a string from the user. The Xlib command *XMapWindow()* will ultimately

make the widget visible. As mentioned previously, the programmer supplies a notify procedure for use following a Return, Enter or Line Feed entry. Usually this procedure will perform error checking on the string. If the string agrees with the formatting requirements, program flow continues. If an incorrect format for the string is detected, the keyboard focus should be returned to the widget and an error message displayed for the user.

Most of the functionality of this and other widgets comes from their initial structure and background functions. Use of the widget involves creating an instance of the widget, mapping it to the screen and providing a notify function for the widget to direct program control.

Now the *GetText* widget offers a similar level of functionality as SunView's *PANEL_TEXT* object gives to the ESKAPE/CF program. SunView's *panel_create_item()* command generates a text input object such as that to obtain the cue name in Figure 4. Unlike the *GetText* widget, a prompting label can be designated as shown below in the command from the ESKAPE/CF edit cue panel.

```
name_item = panel_create_item( data_panel, PANEL_TEXT,
    PANEL_LABEL_X,                  ATTR_COL( GET_DATA_ITEM_X ),
    PANEL_LABEL_Y,                  ATTR_ROW( GET_DATA_ITEM_Y ),
    PANEL_LABEL_FONT,               bold,
    PANEL_VALUE,                    new_data.name,
    PANEL_LABEL_STRING,             "Cue name: ",
    PANEL_VALUE_STORED_LENGTH, MAX_CUE_NAME - 1,
    PANEL_VALUE_DISPLAY_LENGTH, MAX_CUE_NAME - 1,
    PANEL_NOTIFY_PROC,              validate_cue_name,
    0 );
```

Figure 4. The ESKAPE/CF Edit Cue Screen

The following command creates a widget to capture a cue's name in the XESKAPE/CF version shown in Figure 5.

```
get_cue_name = CreateGetText( display, cue_data_panel,
                              GET_CUE_NAME_X,
                              GET_CUE_NAME_Y,
                              GET_CUE_NAME_WIDTH,
                              SINGLE_LINE_HIGH,
                              black, white,
                              font_struct->fid,
                              MAX_CUE_NAME,
                              validate_cue_name );
```

2. Static Text

XESKAPE/CF uses two methods to draw static text in a window. The Xlib command *XDrawString()* will draw a string inside the designated window according to that window's graphic context settings. However, if the window is temporarily obscured, any string displayed using this command must be redrawn. The other method uses the *DisplayText* widget. The *DisplayText* widget is actually a subset of the *GetText* widget. It uses a similar data structure but only requires functions to create the widget and set its string value. Details of the widget's operation follow those of the *GetText* object.

As an example of XESKAPE/CF's use of the *DisplayText* widget, the command creating the first line of text in the edit panel of Figure 5 is shown below.

```
edit_cue_prompt_msg = CreateDisplayText( display, cue_data_panel,
                                         GET_DATA_TITLE_X,
                                         GET_DATA_TITLE_Y,
                                         400, 20,
                                         black, white,
                                         plain_font_struct->fid,
                                         "Please enter required cue data..." );
```

Figure 5. The XESKAPE/CF Edit Cue Screen

Similarly, in order for the same message to be displayed in ESKAPE/CF's edit panel of Figure 4, the SunView command requires only the panel name, text position, font style, item variable label and the string itself.

```
panel_create_item( data_panel, PANEL_MESSAGE,
    PANEL_LABEL_X,          ATTR_COL( GET_DATA_TITLE_X ),
    PANEL_LABEL_Y,          ATTR_ROW( GET_DATA_TITLE_Y ),
    PANEL_LABEL_FONT,       bold,
    PANEL_VALUE,            new_data.name,
    PANEL_LABEL_STRING,     "Please enter required data ...",
    0 );
```

3. Buttons

Users of the ESKAPE/CF or XESKAPE/CF control the applications via push buttons. For the XESKAPE/CF program, a push button is a window that responds to *ButtonPress* events originating within its border.

Push buttons provide another example of the of a locally defined Xlib widget. As with the *GetText* widget, the buttons exist in an array of button-type structures. The structure follows Johnson and Reichard's original design for *AppButton* or application buttons (Johnson and Reichard 1990). Their design provided the backbone to all widgets for the XESKAPE/CF program. XESKAPE/CF incorporates *AppButton* into the local widget set in much the same way as the *GetText* and *DisplayText* widgets. The following lists the command to create the Save Cue Data button in Figure 5.

```
save_cue_data_button = CreateButton( display, cue_data_panel, CNTL,
                                     CUE_CNTL_BTN_X3,
                                     CUE_CNTL_BTN_Y,
                                     black, white,
                                     font_struct->fid,
                                     "SAVE CUE DATA",
                                     final_cue_data_check );
```

The command to initialize ESKAPE/CF's SAVE CUE DATA button in Figure 4 is similar to that of previously mentioned SunView items.

```
panel_create_item( data_panel, PANEL_BUTTON,
    PANEL_LABEL_X,          ATTR_COL( 78 ),
    PANEL_LABEL_Y,          ATTR_ROW( BUTTON_ROW ),
    PANEL_LABEL_IMAGE,      panel_button_image( data_panel,
                                "SAVE CUE DATA",
                                STD_BUTTON_WIDTH,
                                0 ),
    PANEL_NOTIFY_PROC,      final_cue_data_check,
    0 );
```

The *panel_button_image()* command is necessary to set up the button's label.

V. CONCLUSIONS AND RECOMMENDATIONS

Software developers have a variety of tools to construct X Window applications. The use of the low level Xlib library is only one means of building a full-fledged X Window program. However, the ESKAPE/CF to XESKAPE/CF program conversion demonstrated the extra programming necessary to achieve the versatility of higher level toolkits. Several factors should be considered prior to committing to the Xlib approach.

A. ENVIRONMENTAL CONSIDERATIONS

The environment under which an application will operate influences the choice of toolkit. Since the higher level tools are usually linked to a particular vendor's implementation of the X Window system, developers limit their application's portability to that vendor's version of the X Window environment.

1. Portability

An application written solely with the Xlib will have the advantage of being portable to any standard X Window implementation. If the program must work across different window management systems, then Xlib becomes a viable alternative. While applications written under either Open Look or Motif will run on the other's respective platforms, the look and feel of the program will be different (Padovano 1991).

2. Interface Consistency

Since X Window does not support a GUI standard, the window manager and other clients must bear the burden of maintaining graphical interface consistency. An application written in Xlib will provide the best chance at standardization across platforms. All graphical calls will be made to the standard X Window graphic primitives and therefore will appear identical on different systems. Writing different versions of the same application under each of the high-level toolkits will produce different interfaces in each

case. The high-level toolkits' widget sets define the appearance of the application's graphical objects.

B. APPLICATION CONSIDERATIONS

Other factors to consider before planning an X Window application include determining any memory constraints of hardware platforms and evaluating the level of programmer experience.

1. Application Size

Regardless of tool used to develop a program, as much as 50% of the code may consist of graphical interface routines (Yee 1991). Applications using high-level toolkits can grow even larger due to the inclusion of widget set libraries within the executable code. While many of a toolkit's library routines may be required, many only contribute to unnecessary overhead.

The XESKAPE/CF program requires more user-generated interface code than ESKAPE/CF but it is inherently more efficient and compact in the compiled form. The lack of sophistication within Xlib forced the creation of additional interface code since most widgets had to be built from scratch. However, the compiled program becomes streamlined and efficient due to the absence of unused toolkit library routines.

2. Programming Experience

The difficulty in using either high-level toolkits or the Xlib library is roughly equivalent. While the high-level toolkits offer a number of predefined graphical objects, the initial learning curve for their use is no less significant than with Xlib. To add to their complexity, the toolkits do not remove the programmer from the Xlib level entirely. Since most higher level toolkits are built upon a Xlib foundation, a programmer must have the ability to interface with the low-level routines.

APPENDIX A

THE HISTORY OF X WINDOW

A. INCEPTION OF X WINDOW

The X Window concept emerged from Stanford University's W windowing system developed by Paul Asente (Nye 1990). In 1984 researchers at the Massachusetts Institute of Technology (MIT) required a software tool to aid in debugging applications across a network (Vereen 1991). Later that year, Bob Scheiffler and Jim Gettys developed what would become X Window (Clanton 1991). Their system provided a graphical interface facility independent of the platform on which it ran. In this way more than just characters could be exchanged across a network. The X Window project became known as Project Athena and was spearheaded by both MIT and Digital Equipment Corporation (Nye 1990). By 1985, with additional outside contributions, MIT began distributing the new windowing system known as X Window System Version 11. Many users already recognized Unix as a standard, particularly for character-based operations, but each platform had its own protocol for displaying graphics (Clanton 1991). The X Window system became one of the first interfaces to allow graphical data interchange regardless of hardware platform.

The most significant misunderstanding about X Window is the notion that it must run under the Unix operating system. X Window was designed to be and remains operating system-independent (Vereen 1991 and Vizachero 1991). Most implementations of X Window will be found on top of Unix environments but others can be found on AmigaDOS, MacOS and VMS (Reichard and Johnson 1991).

B. THE X CONSORTIUM

After X Window's initial conception, MIT controlled the system's evolution and freely distributed the software. In 1987 nine major computer companies jointly funded the

new X Consortium to maintain consistency of the windowing system. MIT retains control over X Window evolution while the sponsors and other industry participants contribute to an on-going research process. Additionally, the X Consortium distributes information about X Window and encourages extensions to the system (Christian 1991). Today, the X Consortium consists of hundreds of members, nearly all of them representing companies with an interest in graphical user interface environments.

C. THE ICCCM

One of the main goals, the X Window system is to provide the mechanisms for developing graphical elements. In no way does the X Consortium intend to dictate policy over how to use X Window. One drawback from this approach arises from the endless number of user interface designs possible. Programs must be able to communicate on some common level if they are to be as interchangeable as the windowing system on which they run. The programs must provide inter-operability (Johnson and Reichard 1990). The Inter-Client Communications Conventions Manual (ICCCM) defines the conventions recommended for all X Window applications in order to achieve a minimum level of synergy with each other. The ICCCM attempts to set standards for application-to-application communications and prevent contention over shared resources and window managers.

APPENDIX B

THE STRUCTURE OF X WINDOW

A. GRAPHICAL USER INTERFACES

X Window supports the graphical user interface (GUI), a concept that significantly altered many users' perceptions of computer interaction. The idea for the GUI began with the Xerox Star and gained widespread attention after the introduction of the Macintosh Operating System in 1984 (Clanton 1991). A GUI attempts to imitate real world structures on the computer display by using a collection of visual metaphors. The most common metaphor is the window which represents a "view" of a document, file or control structure. Other metaphors include icons, buttons, slide bars and scroll panels.

The use of a GUI can enhance the user's environment significantly. By allowing multiple windows on a screen at once, the computer user is no longer limited to just the display of several dozen lines of text. Now several processes can be viewed simultaneously and even stacked on top of each other. Each window can be compared to a separate terminal, each with its own process running inside of the window.

X Window is not a GUI by itself. It provides a foundation on which graphical user interfaces are built (Christian 1991). A basic X Window package does not define all of the most commonly known structures associated with a full-fledged GUI. One of Gettys and Scheiffler's original goals in designing X Window was not to make it do everything (Scheiffler, Gettys and Newman 1988 and Johnson and Reichard 1990). By just giving X Window some basic building blocks, users could then extend the system on their own and create the structures they needed. Again, the object of the designers was to "provide mechanism rather than policy" (Johnson and Reichard 1990) to the X Window environment.

B. THE CLIENT-SERVER RELATIONSHIP

The X Window System contains a unique structure for porting of applications across networks. It uses a client-server relationship that is the reverse of the traditional notion of server and client. A server is usually recognized as the centerpiece of a network with several clients accessing it to share resources and processing power. Figure B-1 shows that the X Window arrangement is quite different. The X server is a program running on the user's workstation while the X client is the user's process which may be running on a separate machine.

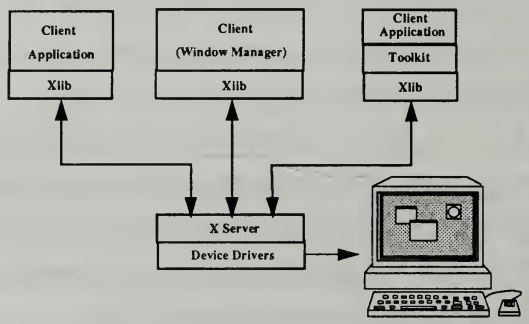

Figure B-1. The Client/Server Relationship (Nye 1990)

1. Server

The X server resides on the local workstation and controls the front-end graphics tasks of the display (Vereen 1991). A display in the X Window environment consists of a keyboard, pointing device and one or more screens (Reichard and Johnson 1991). The server controls input and output from local and remote systems, routing input to the appropriate application while displaying output in the proper windows on the screen. The X server responds to the graphical requests of applications or clients (Christian 1991).

2. Client

The X client is the end-user application program (Vereen 1991). It makes graphical requests of the X server (Christian 1991). The program can run on a separate hardware platform and only has to communicate with the X server via X Window protocols. These protocols enable the client to communicate to the server how it desires its output to be arranged on the screen without having to send detailed code that actually draws it. A client can also access multiple servers simultaneously and therefore appear to run on more than one workstation.

The X server and client can reside on the same machine just as easily as on separate platforms. The important distinction between the client and server is that the server program is built for an individual hardware platform to generate graphics according to its standards. The client uses the same set of standard X Window protocols to express its requests to a server. As a result, X clients have access to a multitude of hardware platforms without the need for code modifications.

C. THE WINDOW MANAGER

The user-interface for a X Window system is provided by the window manager (Padovano 1991). The window manager presents an X server with consistent features such as moving, resizing and stacking of windows, the look of title bars and the appearance of borders and icons (Clanton 1991). The window manager will always be a special client of an X Window system. It handles many generic functions for the clients that would otherwise be bogged down with the extra overhead.

The *twm* window manager comes bundled with the basic X Window Revision 11 package from MIT. It is intended to be a sample window manager and is therefore very simple in its appearance and function (Nye 1990). Even with its simplicity, *twm* and its variants are widely used.

The two most commonly used window managers are AT&T's Open Look Window Manager and Open Software Foundation's (OSF) Motif Window Manager (Padovano 1991). Each system has its own unique look and feel but both provide similar functions. Open Look windows are rounded and have a bulletin-board look (complete with push-pins) while Motif windows appear three dimensional and sculptured (Padovano 1991).

D. THE DESKTOP MANAGER

While the window manager provides the graphical user interface for X Window, the tools for basic file and system management tasks are lacking (Eberle 1991). In order to perform standard file operations as part of the GUI concept, a window system requires another piece of specialized software known as a desktop manager. It simply completes the graphical user interface experience by incorporating point-and-click mouse operations to accomplish tasks such as creation, copying and deletion.

Some desktop managers come bundled with a machine's system software (i.e. Macintosh). With the Unix environment and X Window, a third-party alternative is required to escape the Unix command line (Parrott 1991). Sun Microsystems offers their Open Windows desktop manager to work with the Open Look window manager. Motif does not have an associated desktop manager but it can run in conjunction with other products such as IXI's X.Desktop, Visix Software's Looking Glass or Hewlett Packard's VUE desktop manager.

APPENDIX C

THE GETTEXT WIDGET

```
/************************************ GetText ************************************/
 *
 *   The GetText widget is an object to obtain input from the keyboard.
 *
 *   The structure of some these routines are reproduced from Johnson and Reichard's
 *   Advaced X Window Applications Programming, MIS Press, 1990 and Nye's Xlib Programming
 *   Manual, O'Reilly and Associates, 1990. Although some of the routines have been modified,
 *   their basic structure is a reproduction.
 *
 *   Modifications by: David M. Rust
 */
/********************************** DEFINITIONS **********************************/

#define     BUFSIZE                         50
#define     NULL                            ''
#define     NULL_STRING                     ""
#define     MAX_GET_TEXT_WIDGETS            100
#define     False                           0
#define     True                            1

typedef     int             boolean;

typedef     struct
            {
            Display         *display;       /* the server connection */
            Window          window;         /* the id of the widget's window */
            Window          parent;         /* the id of the widget's parent window */
            GC              gc;             /* graphics context for the widget */
            unsigned long   fore, back;     /* foreground & background settings */
            int             string_length;
            int             (*function) (); /* notify procedure for widget */
            char            string[ BUFSIZE + 1 ];  /* the string to be displayed in widget */
            }
            GetTextStruct;

/****************************** LOCAL FUNCTIONS ******************************/

            void            QuitX();
            Window          CreateWindow();
            GC              MakeGC();
            Window          CreateGetText();
            void            GetTextExec();
            void            SetGetText();
            boolean         GetTextEvent();
            void            AdvanceGetText();
            void            GetChar();
static      void            RedrawGetText();
static      void            UpdateActiveGetText();

/****************************** GLOBAL VARIABLES ******************************/

GetTextStruct       GetText[ MAX_GET_TEXT_WIDGETS ];
int                 get_text_widgets_used = 0;
```

```
/********************************* QuitX ************************************/
/****************************************************************************/
/*   Gracefully exits from the application by disconnecting from the server.
*/

void
QuitX( display, error_message, error_file )

Display     *display;
char        error_message[], error_file[];

{
    (void) fprintf( stderr, "ERROR: %s%s\n", error_message, error_file );

    XCloseDisplay( display );           /* disconnect from the server */

    exit( 1 );                          /* leave the program */

}
/*--- end QuitX ---*/


/****************************** CreateWindow ********************************/
/****************************************************************************/
/*   A simpler procedure to create windows
*/

Window
CreateWindow( display, parent, x, y, width, height, border, fore, back, events )

Display             *display;
Window              parent;                     /* the window's parent window */
int                 x, y, width, height, border;/* the window's location and dimensions */
unsigned long       fore, back;                 /* foreground & background colors */
long                events;                     /* types of events to be recognized by window */

{
    Window                  window;
    XSetWindowAttributes    attributes;
    unsigned long           attribute_mask;
    Visual                  *visual = CopyFromParent;

    /*---------- Set up window attributes */

    attributes.background_pixel= back;
    attributes.border_pixel = fore;
    attributes.event_mask = events;

    attribute_mask = CWBackPixel | CWBorderPixel | CWEventMask;

    /*----------- Create the window */

    window = XCreateWindow( display,
                            parent,
                            x, y, width, height,    /* location, size */
                            border,
                            CopyFromParent,         /* Depth */
                            InputOutput,            /* window class */
                            visual,
                            attribute_mask,
                            &attributes );
```

```
    if ( window == (Window) None )
            {
            QuitX( display, "Error: Could not open window.", "");
            }

    return( window );
}
/*--- end CreateWindow ---*/


/********************************* MakeGC ************************************/
/*****************************************************************************/
/*   A simpler procedure to create gc's.
*/

GC
MakeGC( display, drawable, fore, back )

Display             *display;
Drawable            drawable;               /* the window or pixmap for which to make a gc */
unsigned long       fore, back;

{
    GC              gc;
    XGCValues       gcvalues;

    gcvalues.foreground = fore;
    gcvalues.background = back;

    /*-------------------- create the gc */

    gc = XCreateGC( display, drawable,
                    ( GCForeground | GCBackground ),
                    &gcvalues );

    if ( gc == 0 )
            {
            QuitX( display, "Error in creating a Graphics Context", "" );
            }

    return( gc );

}
/*--- end MakeGC ---*/
```

```
/********************************** CreateGetText ********************************/
/*********************************************************************************/

/* Create a GetText widget to obtain keyboard input from the user. Each widget is actually a Window
 * that accepts events from the keyboard and echoes the input back onto the window.
*/

Window
CreateGetText( display, parent, x,y, width, height, fore, back, font_id, string_length, function )

Display             *display;
Window              parent;
int                 x, y, width, height;
unsigned long       fore, back;
Font                font_id;
int                 string_length;
int                 (*function)();

{
    Window          w;
    GC              gc;

    /*---------------------- Find a slot */

    if ( get_text_widgets_used < ( MAX_GET_TEXT_WIDGETS - 1 ) )
            {
            w = CreateWindow( display, parent,
                              x, y, width, height, 0,
                              fore, back,
                              ExposureMask | ButtonPressMask | KeyPressMask | EnterWindowMask |
                              LeaveWindowMask );

            /*---------------- Create a GC and assign font */

            gc = MakeGC( display, w, fore, back );
            XSetFont( display, gc, font_id );

            /*---------------- Store values */

            GetText[ get_text_widgets_used ].display= display;
            GetText[ get_text_widgets_used ].window= w;
            GetText[ get_text_widgets_used ].parent= parent;
            GetText[ get_text_widgets_used ].gc= gc;
            GetText[ get_text_widgets_used ].fore= fore;
            GetText[ get_text_widgets_used ].back= back;
            GetText[ get_text_widgets_used ].function= function;
            GetText[ get_text_widgets_used ].string_length= string_length;
            strcpy( GetText[ get_text_widgets_used ].string, NULL_STRING );

            XFlush( display );

            /*------------------- Increment slot pointer */

            get_text_widgets_used++;

            return( w );
            }
    else
            return( 0 );
}
/*--- end CreateGetText ---*/
```

```
/************************************ FindGetTextWidget ***************************/
/**********************************************************************************/
/*   Given the window identifier for a GetText structure, this routines finds the array element containing it.
*/

int
FindGetTextWidget( display, window )

Display                 *display;
Window                  window;
{
    int         which_widget;

    for ( which_widget = 0; which_widget < get_text_widgets_used; which_widget++ )
                {
                if (( window == GetText[ which_widget ].window ) &&
                 ( display == GetText[ which_widget ].display ))
                        return( which_widget );
                }
    return( -1 );
}
/*--- end FindGetTextWidget ---*/



/************************************ GetTextExec *********************************/
/**********************************************************************************/
/*   Executes the notify function for the GetText widget. Called from GetChar().
*/

void
GetTextExec( display, which_get_text )
Display                 *display;
int                     which_get_text;
{
    ( GetText[ which_get_text ].function )( display, GetText[ which_get_text ].string, which_get_text );

    return;
}
/*--- end GetTextExec ---*/
```

```
/************************************ SetGetText ********************************/
/********************************************************************************/
/*   Allows you to change the value of the string in a GetText structure.
*/

void
SetGetText( display, get_text_widget, string )
Display             *display;
Window              get_text_widget;
char                string[ BUFSIZE - 1 ];
{
    int             which_get_text;

    /*------------------------ Find the right window */

    which_get_text = FindGetTextWidget( display, get_text_widget );

    if ( which_get_text >=0 )
     {
     /*------------------------ Clear the old value */

     XClearWindow( display, GetText[ which_get_text ].window );

          /*------------------------ Set the string */

          strcpy( GetText[ which_get_text ].string, string );

          RedrawGetText( display, which_get_text );
     }
    return;
}
/*--- end SetGetText ---*/


/************************************ GetTextEvent ******************************/
/********************************************************************************/
/*   GetTetxEvent is used by the EventLoop to update a GetText structure based on the event.
*/

boolean
GetTextEvent( display, event )
Display             *display;
XEvent              *event;
{
    int             which_widget;

    switch ( event->type )
          {
          case Expose:
               which_widget = FindGetTextWidget( display, event->xexpose.window );

               if ( which_widget >= 0 )
                    {
                    RedrawGetText( display, which_widget );
                    return ( True );
                    }
               break;
```

```
        case KeyPress:
                which_widget = FindGetTextWidget( display, event->xkey.window );

                if ( which_widget >= 0 )
                        {
                        GetChar( display, event, which_widget );
                        return ( True );
                        }
                break;
        case EnterNotify:
                which_widget = FindGetTextWidget( display, event->xcrossing.window );

                if ( which_widget >= 0 )
                        {
                        AdvanceGetText( display, event->xcrossing.window );
                        return ( True );
                        }
                break;
        case LeaveNotify:
                which_widget = FindGetTextWidget( display, event->xcrossing.window );

                if ( which_widget >= 0 )
                        {
                        XClearWindow( display, event->xcrossing.window );
                        RedrawGetText( display, which_widget );
                        return ( True );
                        }
                break;
        case ButtonPress:
                which_widget = FindGetTextWidget( display, event->xbutton.window );

                if ( which_widget >= 0 )
                        {
                        AdvanceGetText( display, event->xbutton.window );
                        return ( True );
                        }
                break;
        default:
                break;
        }

    XFlush( display );

    return ( False );
}
/*--- end GetTextEvent ---*/
```

```
/********************************** AdvanceGetText ************************/
/*****************************************************************************/
/*   Sets the keyboard focus to the selected window for a GetText structure.
*/

void
AdvanceGetText( display, get_text_widget )
Display             *display;
Window              get_text_widget;
{
    int             which_get_text;


    which_get_text = FindGetTextWidget( display, get_text_widget );

    UpdateActiveGetText( display, which_get_text );

    XSetInputFocus( display, GetText[ which_get_text ].window, RevertToParent, CurrentTime );

    return;
}
/*--- end AdvanceGetText ---*/


/************************************ GetChar *********************************/
/*****************************************************************************/
/*   GetChar is used by GetTextEvent to handle a keypress inside a GetText widget. It gets the keystroke
/* and determines what to do with it: add it to the string, ignore it, delete a char or call the GetText function.
*/

void
GetChar( display, event, which_get_text )
Display             *display;
XEvent              *event;
int                 which_get_text;         /* the GetText struct to operate on */

{
    int                     count;
    char                    *buffer[1];
    int                     bufsize = BUFSIZE;
    KeySym                  keysym;
    XComposeStatus          compose;
    int                     length;

    count = XLookupString( event, buffer, bufsize, &keysym, &compose );

    /*--------------------------- if Enter, Return or LineFeed call the GetText function */

    if (( keysym == XK_Return ) || ( keysym == XK_KP_Enter ) || ( keysym == XK_Linefeed ))
            {
            XClearWindow( display, GetText[ which_get_text ].window );

            RedrawGetText( display, which_get_text );

            GetTextExec( display, which_get_text );

            return;
            }
```

```
    /*------------------------- if a regular key, add it to the string unless > string_length */

    else if ((( keysym >= XK_KP_Space ) && ( keysym <= XK_KP_9 )) ||
             (( keysym >= XK_space ) && ( keysym <= XK_asciitilde )))
             {
             if (( strlen( GetText[ which_get_text ].string ) + strlen( buffer )) >=
                                                    GetText[ which_get_text ].string_length )
                      XBell( display, 100 );
             else
                      strcat( GetText[ which_get_text ].string, buffer );
             }

    /*--------------------------- skip if key is a modifier key */

    else if (( keysym >= XK_Shift_L ) && ( keysym <= XK_Hyper_R ))
             ;

    /*--------------------------- if Backspace or Delete, remove one char */

    else if (( keysym == XK_BackSpace ) || ( keysym == XK_Delete ))
             {
             if ( strlen( GetText[ which_get_text ].string ) > 0 )
                      {
                      length = strlen( GetText[ which_get_text ].string );
                      GetText[ which_get_text ].string[ length - 1 ] = NULL;
                      XClearWindow( GetText[which_get_text].display,
                                    GetText[which_get_text].window );
                      }
             else
                      XBell( display, 100 );
             }

    /*--------------------------- if any other key, skip it and beep */

    else
             XBell( display, 100 );

    UpdateActiveGetText( display, which_get_text );
    return;
    }
/*--- end GetChar ---*/
```

```
/************************************ RedrawGetText ****************************/
/*******************************************************************************/
/*    Redraws the string in the GetText window. Used by GetChar() and the EventLoop when a Refresh
/* is required.
*/

static
void
RedrawGetText( display, which_get_text )
Display             *display;
int                 which_get_text;
{
    int             font_height;

    font_height = font_struct->ascent + font_struct->descent;

    XDrawString( display, GetText[ which_get_text ].window, GetText[ which_get_text ].gc,
                    2, font_height,
                    GetText[ which_get_text ].string, strlen( GetText[ which_get_text ].string ) );
    return;
}
/*--- end RedrawGetText ---*/


/********************************** UpdateActiveGetText ****************************/
/*******************************************************************************/
/*    Redraws the string in the GetText window but with a caret afterwards to indicate
 * that this item is the active widget. Used by GetChar().
*/

static
void
UpdateActiveGetText( display, which_get_text )
Display             *display;
int                 which_get_text;
{
    int             font_height,
                    string_length,
                    i;

    font_height = font_struct->ascent + font_struct->descent;
    string_length = XTextWidth( font_struct, GetText[ which_get_text ].string,
                                    strlen( GetText[ which_get_text ].string ) );

    XClearWindow( display, GetText[ which_get_text ].window );

    XDrawString( display, GetText[ which_get_text ].window, GetText[ which_get_text ].gc,
                    2, font_height,
                    GetText[ which_get_text ].string, strlen( GetText[ which_get_text ].string ) );

/*------------------------------------------------- Draw the caret, a small, filledtriangle*/

    for ( i=8; i>4; i-- )
     {
     XDrawLine( display, GetText[ which_get_text ].window, GetText[ which_get_text ].gc,
                    string_length + (9-i) + 2, font_height - (9-i) + 4,
                    string_length + (i-1) + 2, font_height - (9-i) + 4 );
     }
    return;
}
/*--- end UpdateActiveGetText ---*/
```

APPENDIX D

GLOSSARY OF TERMS

Bitmap A two dimensional array of bits in which each array element corresponds to a single pixel on a display screen. A monochrome monitor requires only a single bit per pixel. Gray scale and color monitors have multiple bits associated with each pixel in order to define the multiple shades and colors.

Client An application program of a window system server. The client makes graphic requests of the server and responds to a server's event messages.

Desktop Manager A client that provides a virtual desktop with a more intuitive file management interface. It serves to replace much of an operating system's command line structure with graphical metaphors such as icons and buttons.

Display A collection of one or more screens, a keyboard and a pointing device all driven by a single server.

Event The means by which clients are notified of display input or the side effects of earlier requests made of the server. Events are linked to a window and are only sent if the client specifically asked for that type of event.

Event Mask A structure used to define which events are requested for a window.

Expose Event A signal sent to a client notifying it that a window has been initially mapped or is no longer obscured by another window. The event serves as an indication that all or part of a window needs to be redrawn.

Graphics Context A resource containing information for graphics output. A graphics context defines characteristics such as foreground and background pixel color and line widths.

Keyboard Focus The window receiving input from the keyboard. It is possible for keyboard events to go to a window which does not have the pointer in it. The root window holds the keyboard focus by default.

Mapping Mapping a window makes it capable of being displayed. The server will not immediately draw the window onto the screen. The window is only made eligible for display provided that its parent is already mapped and that the window is not obscured by another window.

Root Window The background of a screen. The root window is always present and has no parent.

Screen A virtual, independent display device. Screens can refer to the same or different physical monitors. Several screens can be associated with a single display.

Server A server controls one keyboard and pointing device and one or more screens of a display. It translates graphics requests from a client into screen commands for the hardware platform. The server also accepts input from the display and translates it into events for a client.

Toolkit Library of data structures and functions that is built on top of lower level libraries. In X Window, a toolkit usually refers to a widget set and is built on top of Xlib to extend the basic library of functions.

Widget An abstraction for an interface object. Widgets are software structures that can be created and manipulated. Push buttons and scroll panels are examples of widgets.

Window A subdivision of a bitmapped screen. A window is usually rectangular serves as a virtual screen. Windows make viewing and controlling different tasks on the same screen at the same time less confusing.

Window Manager A special client that maintains authority over the manipulation of windows on the screen.

Windowing System A collection of programming tools and routines for creating, manipulating and interacting with windows on a computer platform.

LIST OF REFERENCES

Brown, Laure, "Graphical User Interfaces: A Developer's Quandary," *Patricia Seybold's Unix in the Office*, v.4, pp. 1-11, August 1989.

Christian, Kaare, "The X Window System: A Universal Graphical Interface," *PC Magazine*, v.10, pp. 323, May 1991.

Clanton, Chuck, "An Introduction to the X-Window System: X-Window server provides standard interface to graphics hardware," *Microprocessor Report*, v.5, pp. 7-9, 20 March 1991.

Conner, James, *Knowledge Acquisition Tool for Expert Systems Using Cognitive Feedback Techniques*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1991.

Eberle, Andrew E. and Jenkins, Avery L., "Desktop Managers Take GUIs to their Golden Age," *Digital Review*, v.8, pp. 25-27, March 1991.

Johnson, E. F. and Reichard, K., *Advanced X Window Applications Programming: The Basics and Beyond*, pp. 1-338, Management Information Source, Inc., 1990.

Johnson, Eric F. and Reichard, Kevin, "X Window Programming, Part 5: X Toolkit Programming," *C Users Journal*, v.9, pp. 59-65, November 1991.

Lainhart, Todd, "Intrinsics of the X Toolkit: A Toolkit for Configurig Your User Interface," *Dr. Dobb's Journal*, v.16, pp. 94-105, February 1991.

Millikin, Michael D., "Sun's OpenWindows: The Workgroup Macintosh?," *Patricia Seybold's Unix in the Office*, v.5, pp. 1-12, October 1990.

Nye, Adrian, *Xlib Programming Manual*, 2nd ed., pp. 1-298, O'Reilly & Associates, Inc., 1990.

Padovano, Michael, "Motif and Open Look: Two Views on Managing Windows," *Systems Integration*, v.24, pp. 25, June 1991.

Parrot, Michael, "UNIX Desktop Managers," *MIDRANGE Systems*, v.4, pp.44-46, 15 October 1991.

Reichard, Kevin and Johnson, Eric F., "The Joy of X," *UNIX Review*, v.9, pp. 95-97, January 1991.

Sheifler, Robert W., Gettys, James, and Newman, Ron, *X Window System: C Library and Protocol*, Digital Press, 1988.

Simpson, David, "Make Unix Easier to Use," *Systems Integration*, v.24, pp. 48-50, August 1991.

Thareja, Ashok K. and Ramachandran, Sridhar, "Migration from ASCII to X," *UNIX Review*, v.9, pp. 35-38, November 1991.

Vereen, Lindsey, "Window of Opportunity: The X Window system may give users of PC networks the best of all possible worlds," *LAN Magazine*, v.6, pp. 123-128, November 1991.

Vizachero, Rick, "X Marks the Spot, But It's Not On the Mark," *Government Computer News*, v.10, pp. 98, 14 October 1991.

Yee, Diana A., "GUI Programming Grows Easier," *UNIX Review*, v.9, pp. 121-125, October 1991.

BIBLIOGRAPHY

Gehani, Narain, *C: An Advanced Introduction (ANSI C Edition)*, Computer Science Press, Inc., 1988.

Miller, Lawrence H., *Programming in C*, John Wiley & Sons, Inc., 1986.

Solbourne Computer, Inc., *X Window Programming Guides: Xlib - C Language X Interface (X Version 11, Release 4)*, 1990.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Professor Tung Bui, Code AS/Bd Department of Administrative Science Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Professor Kishore Sengupta, Code AS/Se Department of Administrative Science Naval Postgraduate School Monterey, CA 93943-5000	1
5.	LT David M. Rust 5431 Beechnut Street Houston, TX 77096-1215	2

Thesis
R9215 Rust
c.1 An introduction to X
Window application devel-
opment.